A Preliminary Analysis of the MPI Queue Characterisitics of Several Applications

Ron Brightwell Sue Goudy Keith Underwood
Sandia National Laboratories*
PO Box 5800
Albuquerque, NM 87185-1110

E-mail: {rbbrigh,spgoudy,kdunder}@sandia.gov

Abstract

Understanding the message passing behavior and network resource usage of distributed-memory message-passing parallel applications is critical to achieving high performance and scalability. While much research has focused on how applications use critical compute related resources, relatively little attention has been devoted to characterizing the usage of network resources, specifically those needed by the network interface. This paper discusses the importance of understanding network interface resource usage requirements for parallel applications and describes an initial attempt to gather network resource usage data for several real-world codes. The results show widely varying usage patterns between processes in the same parallel job and indicate that resource requirements can change dramatically as process counts increase and input data changes. This suggests that general network resource management strategies may not be widely applicable, and that adaptive strategies or more fine-grained controls may be necessary for environments where network interface resources are severely constrained.

1. Introduction

There are many challenges to running an application on a large-scale, distributed-memory massively parallel processing (MPP) machine. Much attention has been directed toward understanding the performance and scalability of applications. This focus has been on understanding and characterizing resource usage including host processors, memory, and, to a limited extent, the network. The goal of most performance analysis tools is to provide the insight necessary to insure that an application is using important resources to its maximum benefit. Every effort is made to find the appropriate strategies for management of processor cycles, the memory subsystem, and the network. The performance and scalability of an application is largely determined by how well these resources can be used as the scale of the system increases.

While significant effort has been directed toward understanding the usage of host processors and memory, there is relatively little effort aimed at understanding and characterizing network resource usage, and network interface resources in particular. Networks are typically measured in terms of micro-benchmarks that demonstrate the maximum performance potential in idealized situations. Although this set of micro-benchmarks has been extended to enable measurements of network performance using typical application scenarios, there is currently little understanding or published researched of what typical scenarios really are.

Even as the struggle to understand the behavior of real-world applications continues, networks are requiring greater resources. New bus technologies, such as PCI Express and HyperTransport, have enabled lower latencies than were previously possible, and advanced signaling technology has led to a significant increase in network bandwidth. To leverage performance increases, network interfaces have increased in processing power and memory capacity. As network interfaces become more complex and the number and type of resources associated with the network continues to increase, the general lack of understanding also increases. In order to address this problem, the amount of effort directed toward understanding network resource usage will need to be similar to that currently being put into gathering, analyzing, and obtaining insight from host processing and memory resources.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This paper presents an initial analysis of the network interface processing and memory requirements of several real-world applications. There are two important impacts of this analysis. First, this type of data can be beneficial from an application performance standpoint. The application may be modified to better match a fixed resource management scheme, or, conversely, a more efficient management scheme can be employed to meet the needs of the application. Second, this type of data can be extremely beneficial for designing future networks and network interfaces.

The rest of the paper is organized as follows. In the next section, we provide additional background information, which is followed in Section 3 by a description of our approach to characterizing network resource usage. In Section 4, we describe the platform from which our data has been collected and provide details about the applications that were used. Results and analysis of the collected data are presented in Section 5. The important conclusions of this study are discussed in Section 6. Section 7 discusses this work in the context of similar studies, and Section 8 provides an overview of future work on this topic.

2. Background

2.1. Placement and Management of Network Resources

Network resources and their associated management strategies have continued to evolve throughout the existence of distributed memory MPP platforms. In early MPPs, such as the Intel Paragon[13], the operating system allocated and managed network resources. Accessing the network required an application to invoke the operating system. The operating system managed buffer space, implemented flow control protocols, and allocated other resources (such as message descriptor handles for identifying asynchronous transfers). When a resource was exhausted, the operating system had to allocate new resources or recover used ones. For example, an application using the NX message passing interface had a (fixed) maximum number of outstanding message receive descriptors. A process that tried to exceed this limit would block until a transfer completed and a previously allocated descriptor was freed.

Research on these early platforms led to new strategies for dealing with network resources. For example, a fundamental part of the Puma [19] lightweight kernel research at Sandia National Laboratories and the University of New Mexico is the Portals network API [6]. Portals moves nearly all of the networking resources into the application's address space and provides building blocks that can be assembled to handle many different types of protocols. By moving resources to the user-level, the application is only constrained by the amount of its memory that it wants to dedicate to networking. It also controls how much processing will be needed to handle messaging requests. This strategy has several benefits. The size of the Puma kernel is fixed and does not change as a process allocates and consumes more network resources, and the complexity of the kernel is significantly reduced.

Recent networking technology has shifted the placement of network resources and requires a reexamination of resource usage and management strategies. Network interfaces now have processor and memory resources dedicated to handling network activities; however, these resources are significantly less capable than the resources on the host. The embedded processors used on current-generation high-performance network interfaces are at least an order of magnitude slower than typical host processors, and the amount of on-board memory is one to four orders of magnitude smaller than host memory.

This arrangement is prevalent in large and extreme scale systems. ASCI Q (over 8000 processors) uses the Quadrics network[14], which handles many networking tasks using a user programmable thread processor on the NIC. Numerous large clusters[2, 1] use the Quadrics or Myrinet network (which also has a processor on the NIC). The Red Storm supercomputer, a joint project between Cray, Inc., and Sandia, has over 10,000 processors[3, 8] and uses a custom network designed by Cray. The network interface for Red Storm has a 500 MHz PowerPC and 384 KB of on-board memory, while the host node has a 2.0 GHz AMD Opteron and 2 GB of main memory. Thus, the network interface is tightly constrained in memory resources and somewhat constrained in processing resources. Even the 65,000 node IBM Blue Gene/L supercomputer must consider the allocation of resources to networking as it uses a similar architecture to the Intel ASCI Red machine[20] currently deployed at Sandia. In this architecture, two processors (700 MHz PowerPC 440 processors on Blue Gene/L and 333 MHz Pentium II processors on ASCI Red) share main memory and share access to the network(s). Network resource usage will significantly impact how application and network processing is divided between the processors.

2.2. MPI Network Resources

Although the high-performance networks in large-scale distributed-memory machines are often used for other traffic (e.g. I/O traffic), MPI is typically the most important service to analyze in terms of resource usage. This importance arises from two factors. First, MPI typically has the biggest influence on the network performance of an application. Second, MPI is often the only service that is primarily controlled by the user rather than the system. That is, MPI is the only network service where the application programmer is largely in charge of behavior at *both* the sender and the

receiver. Other services involve system-level components that provide the opportunity to tightly control resource allocation and management.

Different networks require different levels of host computation to process MPI messages. For example, one network may require a host processor to setup and monitor network DMA activity, while another network may completely decouple the host processor from the network, and avoid any host processor involvement in data transfers. Since characteristics of this type are highly network dependent, we have taken a more general approach to characterizing MPI processor resources by analyzing message queue data.

Conceptually, MPI implementations have two message queues — one that contains a list of outstanding receive requests (the posted receive queue) and one that contains a list of messages that arrived without a posted receive request (the early arrival or unexpected queue). The posted receive queue must be traversed when a message arrives. For most implementations, which represent this queue with a linear list[1], the processing time grows with the length of the queue [21]. Likewise, the unexpected queue must be traversed whenever a receive request is posted. The MPI implementation must atomically check the unexpected queue for a matching message before the request is added to the posted receive queue. Again, traversing this (typically) linear list requires processing resources.

There are several ways in which an MPI implementation can consume memory. Networks typically have a finite number of send and receive requests that can be allocated. In some cases, implementations must use sophisticated credit-based schemes for efficiently managing network transfer requests. Implementations also need to set aside memory for buffering unexpected messages. This memory resource is probably the biggest single concern for any MPI implementor. Since most implementations send short messages eagerly to optimize for latency, situations like an N-to-1 communication pattern can quickly exhaust the buffer space for unexpected messages.

3. Approach

In this section, we describe how we have instrumented the MPICH [12] implementation of MPI to gather information about MPI network resource usage. We chose to instrument MPICH because it is the supported production MPI implementation on our target platform, which is described in detail in the following section. This allowed us to leverage existing application configuration and build environments.

[1]Other data structures (such as hashing) are occasionally used, but many of them can be foiled by applications that wildcard the source and message tag fields in an `MPI_Recv`.

The MPICH implementation has an abstract device interface (ADI) [11] that provides a network transport layer with the functions necessary to implement MPI semantics. In particular, the posted receive queue and unexpected message queue are linked lists that are managed by the ADI code. These linked lists are not usually manipulated by the underlying transport layer, because the ADI abstracts the implementation of these queues.

For example, the ADI provides a function call, `MPID_Msg_arrived()`, for the transport layer to use to signal the arrival of a message. This function traverses the posted receive queue to see if there is a matching receive posted. If so, it removes the entry from the queue and proceeds. If not, it enqueues information about the new message in the unexpected queue. In order to measure the average number of times the posted receive queue is searched, we increment a counter when `MPID_Msg_arrived()` is called. Inside this function call, we also increment another counter each time a queue entry is inspected.

This function call was also used to track the number of unexpected and expected messages. At each invocation of the function, a counter is incremented based on the type of message. In order to have more detail about short versus long messages, we traced the code further down into the device-specific transport layer (ch_gm) and inserted counters there.

The unexpected queue must be searched each time an MPI receive is posted. The MPICH ADI function, `MPID_Search_unexpected_queue_and_post()`, searches through the unexpected queue looking for a matching message. If no match is found, the receive is added to the posted receive queue. If a match is found, the unexpected message is dequeued and the receive is processed. This function calls another ADI function that searches the unexpected queue. We simply increment a counter each time this function is called, and increment another counter each time an unexpected queue entry is inspected.

We also profiled the queue management utility functions in the MPICH ADI to keep track of maximum queue length (for each queue). Each time an entry is enqueued, we increment a length counter associated with the queue. Likewise, this counter is decremented each time an entry is dequeued. Each time a new entry is enqueued, we inspect the length counter to maintain its maximum value.

In order to allow applications to access these counters and maximum values, they were implemented as global variables. This approach allows them to be initialized without an explicit function call and allows them to be exported to the application easily. The MPICH ADI is not multithreaded, so the global values are only manipulated by a single thread of execution.

The data was collected through the MPI profiling in-

terface and written to a file. We defined our own `MPI_Finalize()` routine to record the values and gather them to rank 0, which opens a text file and writes them out for each rank. This eliminates the need to modify applications. We simply re-link the code with the profiling code and the instrumented MPI library.

The overhead of instrumenting MPICH this way is negligible. The additional computation needed for this instrumentation is insignificant, especially for unexpected messages, which are already in the low-performance path. For a posted message, the computation and logic operations are performed after the message has been received, so the additional computation does not impact the transfer of the data. To reduce variability, we ran each test four times and report the average of the runs. Each run was made on the same set of compute nodes for each of the different processor counts.

Because the determination of expected and unexpected messages and the implementation of MPI message queues are specific to each MPI implementation, and possibly specific to each transport layer within an MPI implementation, general instrumentation strategies, such as those used for performance analysis, are not sufficient. There is an ongoing effort to standardize some of this information in a way that application developers as well as tool implementors can use, which we describe in Section 7.

4. Platform and Applications

All tests were run on the Vplant machine at Sandia National Laboratories. Vplant is a Linux cluster with approximately 320 compute nodes composed of Intel Pentium-III and Pentium-4 processors. These experiments were run on dual-processor Pentium-4 Xeon nodes running at 2.0 GHz. Each node has 1 GB of main memory and a Myrinet-2000 [4] network interface. The nodes are connected in a Clos topology. Vplant was running a Linux 2.4.18 kernel, GM version 1.6.4, and MPICH/GM version 1.2.4..11. All of our runs used only one process per node. A number of production applications were evaluated on the experimental platform, including LAMMPS, CTH, and ITS.

4.1. LAMMPS

LAMMPS is a classical molecular dynamics (MD) code designed to simulate systems at the atomic or molecular level[17, 16, 18]. Typical applications include simulations of proteins in solution, liquid-crystals, polymers, zeolites, or simple Lenard-Jones systems. It runs on any parallel platform that supports the MPI message-passing library [2].

This study presents data from the *Bead-Spring Polymer Chains* input deck. This is a simulation of a simple system with molecular bonds. Two types of idealized, 50-length, bead-spring polymer chains using different bead sizes are simulated along with some free monomers. The polymer chains first push off from each other for 10000 timesteps and then equilibrate for 10000 timesteps. The simulated system includes 810 atoms and runs for 20000 timesteps.

[2]This text adapted with permission from `http://www.cs.sandia.gov/~sjplimp/lammps.html`.

4.2. CTH

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. Three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes are available. It uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. CTH is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

CTH has two fundamental modes of operation: with or without adaptive mesh refinement (AMR). Adaptive mesh refinement changes the application properties significantly and is useful for only certain types of input problems. Therefore, we have chosen one AMR problem and one non-AMR problem for analysis. The non-AMR input was the traditional *2 Gas* problem which is simplistic, but provides a comparison with previous studies[5]. The AMR input was a representative production run.

4.3. ITS

The Integrated TIGER Series (ITS) is a suite of codes to perform Monte Carlo solutions of linear time-independent coupled electron/photon radiation transport problems. It can simulate problems with or without the presence of macroscopic electric and magnetic fields in multi-material, multi-dimensional geometries. Individual particles are tracked with independent particle histories. Thus, particle transport is assumed to be a linear process in which individual particles do not interact with each other, or alter the medium in which they transport. The ITS data is from an input deck used in a production run.

5. Results and Analysis

In this section, we provide an analysis of the data by looking at trends. We are less concerned with exact queue resources that a particular application uses, but rather are interested in the relationship between queue resources and various parameters, such as the size of the job, the input

data, the distribution across ranks, and the correlation of real application data to popular benchmarks.

5.1. Unexpected Messages

An unexpected message is a message that arrives before a matching receive has been posted. Unexpected messages can cause a significant amount of performance degradation. Unexpected short messages are typically stored in buffers that are managed by the MPI library and copied into the user buffer once a matching receive is posted. Too many unexpected short messages can cause the MPI library to exhaust the space alloacated to store them. Unexpected long messages are typically not buffered at the receiver. Rather, a rendezvous protocol is used to buffer the message in place at the sender. When a matching receive is posted, the receiver takes steps necessary to transfer the data from the sender. By their nature, unexpected long messages do not realize the full bandwidth performance of the network.

Unexpected messages are considered to be the "slow path" because they remain in the unexpected queue for an indeterminate amount of time. Unexpected short messages also have large memory resource requirements to accommodate the buffering of the entire message at the receiver. Figure 1 shows the proportion of messages that falls into each of four categories: expected long, expected short, unexpected long, and unexpected short. Unexpected short messages are clearly quite common in the real applications that were evaluated while unexpected long messages were very uncommon. ITS demonstrates the worst case behavior with the proportion of unexpected short messages appearing to scale linearly with the number of processes in the job. In all of the cases, it is clear that unexpected messages must be handled quickly and that significant memory will be needed to buffer unexpected short messages.

5.2. Queue Lengths

One of the greatest limitations of most modern network interface hardware is the extremely limited amount of memory on the card. As such, the maximum length of the posted receive queue and the unexpected message queue have significant implications for the feasibility of message offload. Figure 2 shows the maximum search length and maximum overall length for the posted receive and unexpected message queues. The maximum queue length is an indication of the amount of memory resources required to store the queue. These results indicate that, even for relatively small numbers of processors, the maximum length of both the posted queue and the unexpected queue are within the limits of the memory that a modern NIC would support; however, the results also indicate several potential problems. The posted queue length of the AMR version of CTH increases with the number of processes in the job. Similarly, the unexpected queue results for the non-AMR version of CTH and LAMMPS appear to scale linearly. This is potentially more significant since the unexpected queue is an order of magnitude longer than the posted queue.

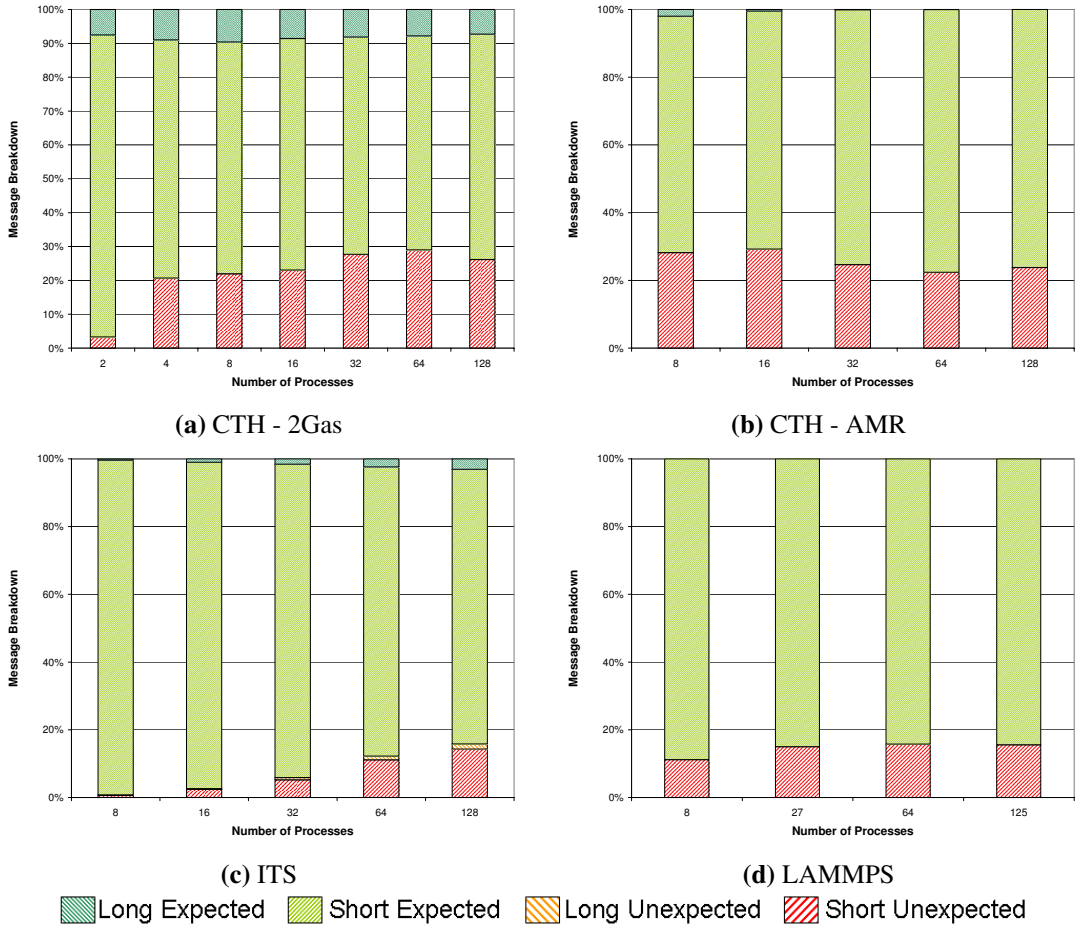
These results also show the disparity between rank 0 and the rest of the ranks in the job. Removing rank 0 from the results, the maximum length of the unexpected queue drops dramatically. This can probably be attributed to the fact that most applications use rank 0 as the root of collective operations. This data indicates that uniform allocation of resources across all ranks may not be optimal.

5.3. Search Length

The search length of a queue is the number of queue entries that are traversed in a given search. While long queues have implications for the amount of memory required for NIC offload, the portion of those queues that are searched has a significant impact on the processing power needed by the NIC. Search length also affects the real latency seen by applications.

Figure 2 shows the maximum search length for these applications. This data reveals many interesting properties. There are several cases for both queues where the maximum search length is the entire length of the queue. This makes sense for the unexpected queue, where a matching entry may not be in the queue when a receive is posted. It is more disconcerting to see that several applications search the entire length of the posted receive queue to find a match. These results seemingly discourage, for example, an offload implementation of the MPI matching semantics where a small portion of the posted receive queue is buffered in NIC memory and the remainder of the queue is traversed by accessing host memory from the NIC.

Figure 3 shows the average search depth of the posted and unexpected queues. Compared to the maximum value, the average traversal of the posted receive queue is extremely small. This is also true for the unexpected queue, except for the ITS application. In this case, the impact of search length at rank 0 is significant. The disparity between maximum search length and average search length is also likely to introduce variability into the execution time of a time step (the time between synchronization points). This type of variability is the key symptom of the "rogue OS effect"[15], which leads to significantly longer applications execution times. This type of variability will also prove to be one of the major limiting factors in scaling from 10,000 to 100,000 nodes.



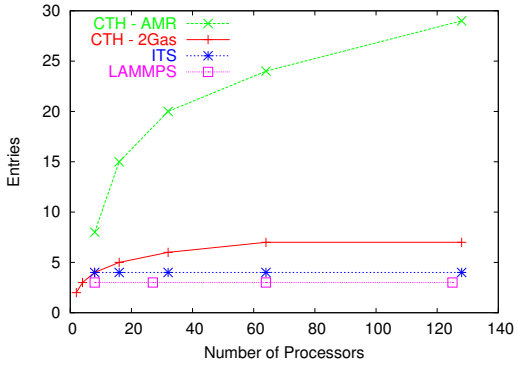
(a) CTH - 2Gas

(b) CTH - AMR

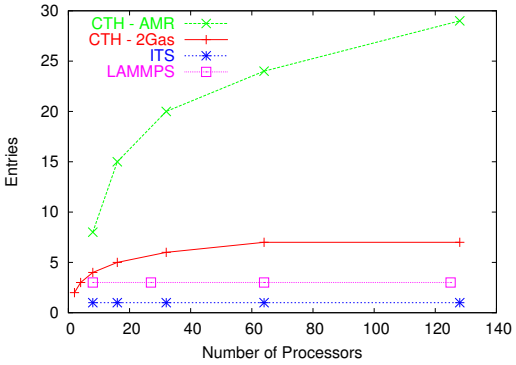
(c) ITS

(d) LAMMPS

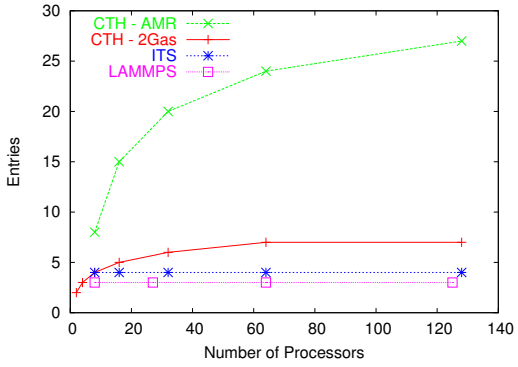
Figure 1. Breakdown of messages by expected/unexpected and long/short properties for applications



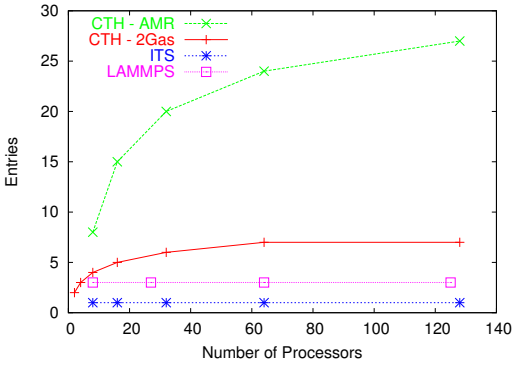
(a) Max posted queue length



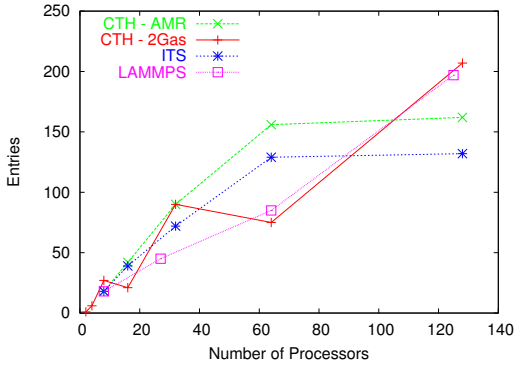
(b) Max posted queue length (no rank 0)



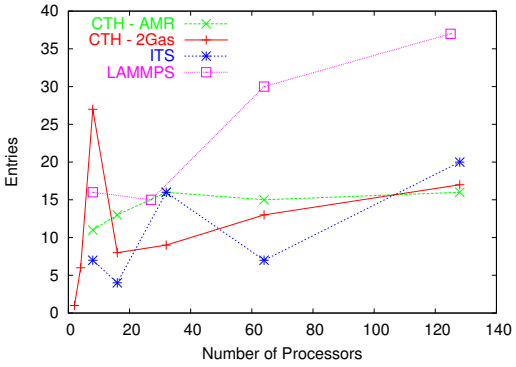
(c) Max posted queue search



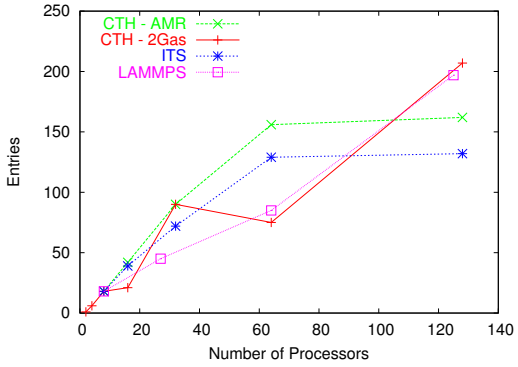
(d) Max posted queue search (no rank 0)



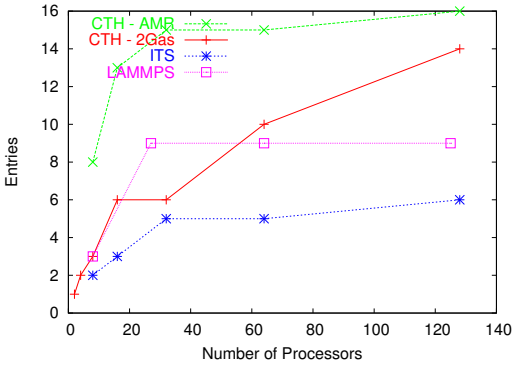
(e) Max unexpected queue length



(f) Max unexpected queue length (no rank 0)

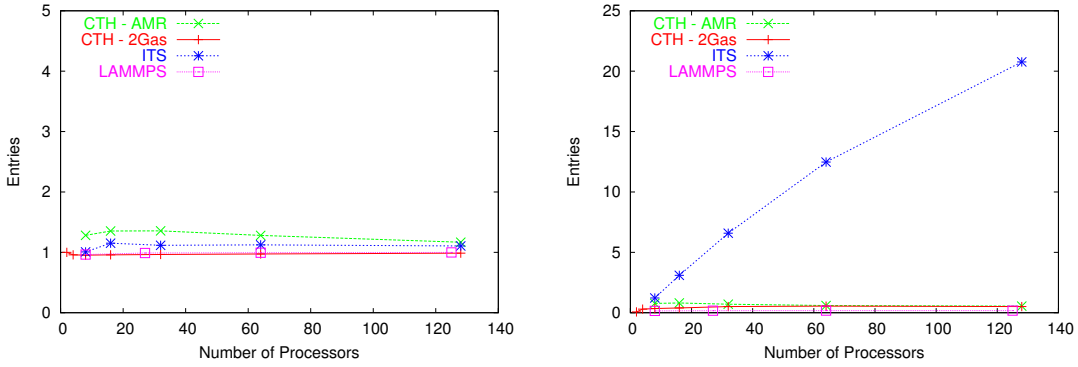


(g) Max unexpected queue search



(h) Max unexpected queue search (no rank 0)

Figure 2. Maximum length and search depths of the posted and unexpected message queue



(a) Average posted queue search

(b) Average unexpected queue search

Figure 3. Average search depth of the posted and unexpected queues

5.4 Comparison to Benchmarks

We now compare the results of these applications with results of the NAS Parallel Benchmark (NPB) suite [9] previously published in [7]. Our experience with real applications thus far shows very little correlation with the behavior of the NPB suite in terms of expected and unexpected messages. In contrast, there is more correlation in the queue behaviors of real applications and the NPB suite. In this case, only the measurements without rank zero are compared. Most of the real applications (like most of the NPB) have small maximum queue lengths, but some are related to the number of nodes. The key difference is that the growth in length for applications appears to be related to $log(P)$ while the growth for the benchmarks appears to be related to P. Similarly, the search behavior of the benchmarks is correlated with, but not identical to, the behavior of the real applications.

6. Conclusions

This paper presented an initial analysis of the message passing behavior of four real application scenarios in terms of the resource usage and resource requirements. The results indicate that, in many cases, certain MPI resource requirements scale with the number of processes. The growth (with process count) of the unexpected queue and the growth in the percentage of messages which are short and unexpected requires that both significant memory and significant processing be dedicated to MPI. Unfortunately, these patterns vary across applications in a way that indicates that a generalized resource management scheme may be inappropriate. Moreover, the usage patterns within one job (specifically for rank 0) vary widely indicating that a single, static resource management scheme may be insufficient even within a single job. These results were compared to the NAS parallel benchmark suite, which was studied using the same analysis techniques. We found that the degree of correlation between the message passing behavior of the NPB suite and Sandia's applications varied based on the parameters being measured. The behavior of the NPB suite appears to be a reasonable first approximation of real applications, but is not truly representative.

7. Related Work

There is a significant amount of work in the area of parallel application performance analysis. However, we know of no work that collects, analyzes, or uses MPI unexpected messages or MPI queue information as a basis to characterize performance, scalability, or network resource usage.

Most performance analysis tools for MPI use the MPI profiling interface to gather message tracing and timing information. Since unexpected messages are not exposed in the MPI programming interface, this information is not available to the profiling layer. Many of the issues with unexpected messages that we described in this paper have motivated work on a portable interface for exposing low-level MPI implementation details, such as unexpected messages, to application developers and performance tool developers. This interface, called PERUSE [10], is currently being explored by a number of organizations in the MPI research community. This work emphasizes the need to be able to capture low-level MPI performance information to assist in characterizing application message passing requirements.

In addition to performance analysis, there is also a significant amount of work that characterizes the message passing behavior of applications and application benchmarks in an attempt to understand or predict how well they will scale. Example of this type of analysis can be found in [22] and [23]. As with performance tools, this analysis does not consider the impact or effect of unexpected messages or queue

lengths, largely because this information is not easily attainable.

8. Future Work

This initial analysis provided answers to some important questions regarding MPI queue behavior for real applications. However, several important questions remain.

We expect that some of our results are platform dependent and that various apsects of a system may have a significant impact on MPI queue usage. For example, the cluster from which our results have been gathered is highly unbalanced in terms of computation to network bandwidth. A platform that provides significantly more network bandwidth may behave much differently. We intend to explore the degree to which these types of system parameters impact MPI queue usage. One system aspect that we intend to explore in depth is the impact of scale. While 128 processors is a significant number, it is well short of the several thousand that we expect for the typical application running on Red Storm.

The approach to instrumentation that we presented is this paper is straightforward. More advanced techniques may be necessary to truly capture the level of detail necessary for performance optimizations or to help develop adaptive strategies for network interface resource allocation and management. In particular, we would like to examine the distribution of MPI queue search data rather than just examining average data over the entire run of an application.

References

[1] http://www.lanl.gov/projects/pink/.

[2] http://www.llnl.gov/linux/mcr/.

[3] R. Alverson. Red Storm. In *Invited Talk, Hot Interconnects 10*, August 2003.

[4] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[5] R. Brightwell, H. E. Fang, and L. Ward. Scalability and performance of CTH on the Computational Plant. In *Proceedings of the Second International Workshop on Cluster-Based Computing*, May 2000.

[6] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[7] R. Brightwell and K. D. Underwood. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[8] W. J. Camp and J. L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[9] R. F. V. der Wijngaart. NAS parallel benchmarks version 2.4. Technical report, October 2002.

[10] R. Dimitrov, A. Skjellum, T. Jones, B. de Supinski, R. Brightwell, C. Janssen, and M. Nochumson. PERUSE: An MPI performance revealing extensions interface. Presented at the Sixth IBM System Scientific Computing User Group, August 2002.

[11] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[13] Intel Corporation. Paragon XP/S product overview, 1991.

[14] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[15] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Identifying and eliminating the performance variability on the ASCI Q machine. In *Proceedings of the 2003 Conference on High Performance Networking and Computing*, November 2003.

[16] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[17] S. J. Plimpton. Lammps web page, July 2003. http://www.cs.sandia.gov/ sjplimp/lammps.html.

[18] S. J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, Mar. 1997.

[19] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[20] S. R. W. Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.

[21] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.

[22] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 27–29, April 2002.

[23] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of the SC99 Conference on High Performance Networking and Computing*, November 1999.